

Mopsa-C: Towards Incorrectness and Termination Verdicts (Competition Contribution)

Marco Milanese¹, Raphaël Monat²(✉)*, Abdelraouf Ouadjaout¹, and Antoine Miné¹

¹ LIP6, Sorbonne Université, F-75005, Paris, France

² Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France

Abstract. We present advances we brought to Mopsa for SV-COMP 2026. Mopsa now supports a backward analysis mode which computes an under-approximation of the weakest liberal precondition for the alarms found during a verification pass, which generates an input harness to reproduce the incorrectness condition. We also added support for termination checking through the introduction of loop counters. With these improvements, Mopsa wins the *SoftwareSystems* category, and ranks third in the *TrueOverall* category.

Keywords: Static Analysis · Abstract Interpretation · Competition on Software Verification · SV-COMP.

1 Verification Approach: the Mopsa Platform

Mopsa is an open-source static analysis platform relying on abstract interpretation [6]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Journault et al. [10] describe the core of Mopsa principles, and Monat [19, Chapter 3] provides an in-depth introduction to Mopsa’s design. The C analysis which we rely on for this competition is based on the work of Ouadjaout and Miné [27]; it proceeds by induction on the syntax, is fully context- and flow-sensitive. This is the fourth time Mopsa participates to SV-COMP [22, 20, 23]. In addition to a sound forward analysis, Mopsa now features a backward under-approximating analysis, and a sound termination analysis.

Backward Analysis. Mopsa now supports a backward analysis based on the abstract interpretation framework presented of Miné [17], Milanese and Miné [14, 15], which computes a liberal precondition. This precondition is used to infer counterexamples for alarms identified during the standard forward over-approximating analysis. The current implementation is experimental and does not support all the abstractions available in the analyzer (e.g., only the interval numeric abstraction is supported, while the forward analysis also supports polyhedra, string length, symbolic rewriting, etc). Nevertheless, the backward analysis supports all primary C constructs, including dynamic memory allocation

* Jury member

(using recency abstraction), low-level memory manipulation through pointers and casts (using a dynamically-typed cell-based memory abstraction), advanced control structures (goto and switch). If enabled, the backward analysis is automatically initiated at the end of the forward analysis whenever alarms are detected. It begins at the end of the program with a postcondition of `false`, and at any program location where a property violation is possible, it combines the current state with those states that *definitely* violate the property. As a result, the analysis ultimately produces a precondition that ensures either divergence or the violation of at least one alarm. Moreover, program inputs (e.g., `__VERIFIER_nondet`) are modeled as program variables, and in general the precondition found involves some constraints on the inputs.

Termination Analysis. We have implemented a forward termination analysis introducing loop counters and checking they are always bounded, following the previous works of Halbwachs [7], Saan et al. [31]. Additionally, we have extended the approach to handle backward gotos.

Other Improvements. We have implemented a preprocessing step ensuring boolean conditions (e.g., within conditionals and loops) are normalized, improving the precision of the analysis. We have refined the iterator handling backward gotos to minimize the number of statements used in the fixpoint iteration. Thanks to the SV-Benchmarks verdicts, we have also been able to fix a stub contract for the `exp` function of the C standard library, and to remove spurious overflow alarms which occurred when using variable length arrays.

2 Software Architecture: the SV-COMP Driver

By default, the C analysis of Mopsa detects all the runtime errors that may happen in the analyzed program, while SV-COMP tasks focus on a specific property at a time. We thus rely on an SV-COMP specific driver. It takes as input the task description (program, property, data model). It uses a sequential portfolio approach, by trying increasingly precise C analyses defined in Mopsa, possibly enabling also the backward version, until the property of interest is (dis)proved, or the most precise analysis is reached (or the resources are exhausted). Each analysis result is postprocessed by the driver to check if the property is proved. An analysis configuration defines the set of domains used and their parameters, allowing control of the precision-efficiency ratio.

Our portfolio is mostly unchanged from last year [23]. We streamlined the unrolling parameters used by default to improve scalability and rely on the autosuggestion hook to detect semantic patterns [23]. We are now using PPLite’s implementation of polyhedra [4], as the overall results seemed better than when using the default implementation of Apron [9].

Violation Verification The analyzer computes a liberal precondition, and to ensure that a value extracted from it constitutes a valid counterexample, it is necessary to verify that the program does not diverge—i.e., gets stuck in an infinite loop without violating the property. To this end, the driver first determines

a concrete input vector by assigning to each input variable the smallest value within its interval of variation. It then substitutes the input functions with these concrete values and compiles the benchmark. Specifically, to detect property violations such as overflows that may not result in program crashes, compilation is performed with C sanitizers enabled, as is done in [29]. After compilation, the program is executed. Whenever a crash is observed, the driver records the crash location and generates a corresponding violation witness. Notably, the crash is deterministic because all input calls are replaced with fixed, concrete values. Additionally, the program notifies the driver each time an input is accessed, enabling the inclusion of such inputs in the final witness. This information is used to determine both the order and the number of times each input function is invoked within the violation trace.

3 Strengths and Weaknesses

Mopsa participated in the following categories, targeting C programs: *ReachSafety*, *MemSafety*, *NoOverflows*, *SoftwareSystems* and *Termination*. An overview of results can be found in the competition report [5]. Mopsa is the best verifier of the *SoftwareSystems* track, which focuses on verifying real software systems. Overall, it also ranks third in the *TrueOverall* category where verifiers only focus on proving programs correct. We show in Figure 1 selected improvements of Mopsa as a whole. Figure 2 summarizes the result of the backward analysis, where we report both validated results and non-validated ones.

Strengths. Compared to last year’s participation Mopsa can now handle also incorrect programs. Thanks to this Mopsa managed to prove 2660 new tasks with false verdict that previously were not accessible due to the analysis method. Additionally, the support for proving program termination allowed Mopsa to prove 1010 tasks to be terminating.

Weaknesses. The implementation of the backward analysis is still experimental and not yet on par with the forward analysis. This is visible in Table 2 in *ReachSafety* where complex properties must be inferred—compared to simpler overflow or memory unsafe accesses—and *SoftwareSystems* where both scalability and precision on extended program sections are necessary. As a future work we would like to add support for more abstractions, as well as further tune the performance-precision tradeoff of this analysis.

Mopsa cannot soundly analyze concurrency-related verification tasks, but we could leverage previous abstract interpretation work targeting those properties [18, 34, 21, 32]. Our termination checker is quite naive and does not support recursive functions. We could reuse more involved—possibly backward—abstract interpretation techniques to prove termination [33, 24].

Our SV-COMP driver currently tries a sequence of increasingly precise configurations: this approach is not efficient, we are planning to develop techniques deciding what would be the best configuration to analyze a given program, following the works of Oh et al. [26], Mansur et al. [12], Wang et al. [35].

Mopsa is currently not generating meaningful correctness witnesses. Given that the witness validation limit has been significantly reduced this year, this is

Subcategory	Prop.	tasks	Mopsa'25	Mopsa'26	Best score, verifier (2026)	
Loops	R	764	386	491	982	aise [11]
Recursive	R	174	60	85	150	Symbiotic [1]
Sequentialized	R	585	4	18	531	CPAchecker [2]
Arrays	M	221	124	179	356	UAutomizer [8]
Juliet	M	3271	2530	3788	4709	CPAchecker [2]
Main	N	1986	2138	2384	2763	UParalizer [3]
BitVectors	T	34	—	10	55	PROTON [25]
ControlFlow	T	281	—	108	441	PROTON [25]
Heap	T	201	—	258	352	PROTON [25]
Other	T	1630	—	1576	2184	PROTON [25]
Busybox	N	65	8	14	14	Mopsa
Other	R	70	12	22	35	ESBMC-kind [13]
Other	M	49	14	18	18	Mopsa
DDL	T	215	—	32	201	UAutomizer [8]
Uthash	T	32	—	36	54	PROTON [25]

Fig. 1. Mopsa’s improvements for selected subcategories from ReachSafety, MemSafety, NoOverflows, Termination and SoftwareSystems track, where we compare the scores reached at SV-COMP 2025 and 2026. Property is either *ReachSafety*, *MemSafety*, *NoOverflow* or *Termination* (previously unsupported). The last columns show the score of Mopsa submitted last year, this year, and the best score reached by a verifier.

Category	false tasks	Validated	Unconfirmed	Best false , verifier	
ReachSafety	3076	137	36	1743	CPAchecker [2]
NoOverflows	3680	1473	92	2847	Symbiotic [1]
MemSafety	2156	1035	7	2042	Symbiotic [1]
SoftwareSystems	1568	15	4	324	Symbiotic [1]

Fig. 2. Mopsa’s backward analysis results for supported categories: number of tasks with a false verdict that Mopsa has been able to produce and validate, or to produce but that was unconfirmed by witness validators.

now a limitation of our approach, encountered for 27 of 9975 tasks where Mopsa finds the task to have result true. We have also noticed that some incorrectness witnesses are not validated. This is surprising given that our witnesses are fully concrete testcases. In the future, we plan to add a sanitizers-based validator, similar to SV-Sanitizer [30]—already used in our verifier—to confirm these tasks.

4 Software Project and Contributors

Mopsa is available on Gitlab [28], and released under an GNU LGPL v3 license. Mopsa was originally developed at LIP6, Sorbonne Université following an ERC Consolidator Grant award to Antoine Miné. Mopsa is now additionally developed in other places, including Inria, ENS, Airbus and Nomadic Labs. The people who improved Mopsa for SV-COMP 2026 are the authors of this paper.

Data-Availability Statement. The exact version of Mopsa and driver that participated to SV-COMP 2026 are available as a Zenodo archive [16].

Funding Statement. This work was supported by France’s Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001. This work was supported by project ANR-22-PECY0005 "Secureval" managed by the French National Research Agency for France 2030. This work was supported by grant agreement ANR-24-CE25-7956-01 RAISIN, and by an Amazon Research Award, Fall 2024.

Acknowledgments. We acknowledge the positive impact of Dagstuhl Seminar #25421 on this work.

Bibliography

- [1] Ayaziová, P., Jonáš, M., Mihalkovič, V., Sedláček, J., Strejček, J.: SYMBIOTIC 11: Predicate abstraction joins the party (competition contribution). In: Proc. TACAS (2), LNCS 16506, Springer (2026)
- [2] Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpcachecker 2.3 with strategy selection - (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 359–364, Springer (2024)
- [3] Barth, M., Dietsch, D., Heizmann, M., Jakobs, M.C.: ULTIMATE PARALIZER: Parallel trace abstraction (competition contribution). In: Proc. TACAS (2), LNCS 16506, Springer (2026)
- [4] Becchi, A., Zaffanella, E.: Pplite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.* **275**, 104620 (2020)
- [5] Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2), LNCS 16506, Springer (2026)
- [6] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 238–252, ACM Press, Los Angeles, California (1977), <https://doi.org/10.1145/512950.512973>
- [7] Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d’un programme. Theses, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I (Mar 1979), URL <https://theses.hal.science/tel-00288805>, universités : Université scientifique et médicale de Grenoble et Institut national polytechnique de Grenoble
- [8] Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER and the abstraction of

- bitwise operations (competition contribution). In: Proc. TACAS (3), pp. 418–423, LNCS 14572, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_31
- [9] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, pp. 661–667, Springer (2009)
- [10] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, pp. 1–18 (2019), https://doi.org/10.1007/978-3-030-41600-3_1
- [11] Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS (3), pp. 199–204, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_12
- [12] Mansur, M.N., Mariano, B., Christakis, M., Navas, J.A., Wüstholtz, V.: Automatically tailoring abstract interpretation to custom usage scenarios. In: CAV (2), Lecture Notes in Computer Science, vol. 12760, pp. 777–800, Springer (2021)
- [13] Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: ESBMC v7.4: Harnessing the power of intervals (competition contribution). In: Proc. TACAS (3), pp. 376–380, LNCS 14572, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_24
- [14] Milanese, M., Miné, A.: Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In: VMCAI, Springer (2024)
- [15] Milanese, M., Miné, A.: Under-approximating memory abstractions. In: Proc. of the 31th International Static Analysis Symposium (SAS’24), Lecture Notes in Computer Science (LNCS), vol. 14995, Springer (Oct 2024), <http://www-apr.lip6.fr/~mine/publi/article-milanese-al-sas24.pdf>
- [16] Milanese, M., Monat, R., Ouadjaout, A., Miné, A.: Mopsa at sv-comp 2026 (Nov 2025), <https://doi.org/10.5281/zenodo.17696794>
- [17] Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Sci. Comput. Program.* **93**, 154–182 (2014), <https://doi.org/10.1016/J.SCICO.2013.09.014>, URL <https://doi.org/10.1016/j.scico.2013.09.014>
- [18] Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI, Lecture Notes in Computer Science, vol. 8318, pp. 39–58, Springer (2014)
- [19] Monat, R.: Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries. Ph.D. thesis, Sorbonne Université, France (2021)
- [20] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: MOPSA-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: Proc. TACAS (3), pp. 387–392, LNCS 14572, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_26

- [21] Monat, R., Miné, A.: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10145, pp. 386–404, Springer (2017), https://doi.org/10.1007/978-3-319-52234-0_21, URL https://doi.org/10.1007/978-3-319-52234-0_21
- [22] Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C: Modular domains and relational abstract interpretation for C programs (competition contribution). In: Proc. TACAS (2), pp. 565–570, LNCS 13994, Springer (2023), https://doi.org/10.1007/978-3-031-30820-8_37
- [23] Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C with trace partitioning and autosuggestions (competition contribution). In: Proc. TACAS (3), pp. 229–235, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_17
- [24] Moussaoui Remil, N., Urban, C.: Termination Resilience Static Analysis. In: VMCAI 2026 - 27th International Conference on Verification, Model Checking, and Abstract Interpretation, Rennes, France (Jan 2026), URL <https://inria.hal.science/hal-05398150>
- [25] Mukhopadhyay, D., Metta, R., Karmarkar, H., Madhukar, K.: PROTON 2.1: Synthesizing ranking functions via fine-tuned locally hosted LLM (competition contribution). In: Proc. TACAS (3), pp. 242–247, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_19
- [26] Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: PLDI, pp. 475–484, ACM (2014)
- [27] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, pp. 223–247 (2020), https://doi.org/10.1007/978-3-030-65474-0_11
- [28] Ouadjaout, A., Monat, R., Miné, A., Journault, M.: Mopsa (2022), URL <https://gitlab.com/mopsa/mopsa-analyzer>
- [29] Saan, S.: Sv-sanitizers in sv-comp 2026 (Oct 2025), <https://doi.org/10.5281/zenodo.17400380>, URL <https://doi.org/10.5281/zenodo.17400380>
- [30] Saan, S.: Sv-sanitizers: Sv-comp wrapper for sanitizers (Dec 2025), URL <https://github.com/sim642/sv-sanitizers>
- [31] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination - (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III, Lecture Notes in Computer Science, vol. 14572, pp. 381–386, Springer

- (2024), https://doi.org/10.1007/978-3-031-57256-2_25, URL https://doi.org/10.1007/978-3-031-57256-2_25
- [32] Suzanne, T., Miné, A.: Relational thread-modular abstract interpretation under relaxed memory models. In: Ryu, S. (ed.) Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings, Lecture Notes in Computer Science, vol. 11275, pp. 109–128, Springer (2018), https://doi.org/10.1007/978-3-030-02768-1_6, URL https://doi.org/10.1007/978-3-030-02768-1_6
- [33] Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings, Lecture Notes in Computer Science, vol. 8723, pp. 302–318, Springer (2014), https://doi.org/10.1007/978-3-319-10936-7_19, URL https://doi.org/10.1007/978-3-319-10936-7_19
- [34] Vojdani, V., Apinis, K., Rötov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblin approach. In: ASE, pp. 391–402, ACM (2016)
- [35] Wang, Z., Yang, L., Chen, M., Bu, Y., Li, Z., Wang, Q., Qin, S., Yi, X., Yin, J.: Parf: Adaptive parameter refining for abstract interpretation. In: ASE, pp. 1082–1093 (2024)

